CS422 - Final Exam

Send it either by email or push it under my door (SC 2110).
Total: 100 (out of 110) points needed for maximum credit. You are *not* allowed to collaborate or discuss these problems with each other. You can use books, lecture notes, computers, coffee, etc.

Problem 1. (25 points) Add parallel assignment to IMP++ in all the semantic approaches discussed in class. For simplicity, consider only parallel assignment of two variables. The statement "`x1,x2 := e1,e2`" first evaluates `e1` and `e2` in parallel and then writes their values to `x1` and `x2`. For example, "`x,y := y,x`" swaps the values of `x` and `y`. By "parallel evaluation" we mean that their evaluations can interleave arbitrarily, which can lead to many different behaviors in IMP++ (since expression evaluation has side effects). All you need to do is to handle paper definitions of parallel assignment in all the semantics styles discussed in class. Where the particular semantics style does not allow you to get all the desired behaviors, say so and give a brief explanation why. It is OK not to get all the behaviors, or even as many as could be obtained, but you need to show that you are aware of it.

Problem 2. (20 points)

- Write a FUN program that calculates all the k-combinations of n elements, that is, all the possibilities to pick k elements from a pool of n elements. It is known that there are $n!/(k! \cdot (n-k)!)$ such combinations; you are supposed to define a function taking two arguments n and k and returning a list of lists of numbers between 1 and n; these lists are regarded as sets (no repetitions and the order of elements does not matter).
- Type the program above by hand using the type inference technique discussed in class. Give enough detail to show that you understand the technique, but not more than that.

Problem 3. (25 points)
`callcc` (see the definition of FUN for a formal definition) grabs the current computation and passes it to its argument, which is expected to be a function. If the passed computation is ever passed a value, the current computation is dissolved and replaced by the passed one. This problem is about what happens if the passed computation is not invoked. `callcc` actually returns to the calling context and the program continues normally from there; for example, both expressions below

```
(callcc (fun k -> k 7)) + 3
(callcc (fun k -> 7)) + 3
```

evaluate to 10. Define a special `callcc`, say `mycallcc`, which never returns to the calling context. With `mycallcc`, the two expressions above evaluate to 10 and 7, respectively. Use K for your definition (paper definition is ok). Can you define `callcc` and `mycallcc` in terms of each other, that is: (1) can you desugar `mycallcc` into `callcc`? and (2) can you desugar `callcc` into `mycallcc`?

Problem 4. (20 points) Add proper exceptions to typed KOOL. In languages like Java, methods explicitly state what types of exceptions they may possibly throw, this way guaranteeing that all explicitly thrown exceptions are eventually caught. Add the same functionality to typed KOOL, and give it both dynamic and static semantics. Paper definition is OK, no need to do it in the tool.

Problem 5. (20 points)
Suppose that you are the designer of a concurrent language and that you'd like to add support for transactional actions into your language by means of an atomicity construct. More precisely, you'd like to add a statement atomic{E}, which evaluates E atomically, that is, without any interference from the other threads (assume that you have already added thread creation and termination to your language). Also, you'd like to add a statement break-atomic, which leaves the atomic block discarding all the computation together with its side effects that have been accumulated while partially executing the atomic block, and unfreezes the rest of the threads. If the atomic block terminates normally, then its side-effects are committed. In other words, atomic{E} freezes the other threads and starts to evaluate E optimistically; if E evaluates normally, then its effects are committed, the other threads are unfrozen, and the computation continues normally; if the evaluation of E encounters a break-atomic statement, then the state of the program is recovered to the moment in which the atomic block was tried as if none of its code has been executed, the atomic block is skipped, the other threads are unfrozen, and the computation continues normally, as if the atomic block was never seen.

This problem has two parts:

1. Sketch K definitions for atomic{E} and break-atomic;

2. Comment on the computational limitations of your definition above, namely on the fact that the rest of the world is frozen in order for an atomic block to stay atomic. Can you do better? Can you let the other threads continue their executions and stop them only if they try to interfere with with the atomic block?